Institut für
Technische Informatik und
Kommunikationsnetze

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Speech Recognition with Hierarchical Codebook Search

Tobias Bühler

Semester Thesis SA-2014-13

Institut für Technische Informatik
und Kommunikationsnetze

Supervisors:

Dr. Beat Pfister and Tofigh Naghibi

Responsible Professor:

Prof. Dr. L. Thiele

May 30, 2014

Contents

Abstract

Speech recognition is today often based on hidden Markov models (HMM) more precisely on continuous density hidden Markov models (CDHMM). That means the observation density is continuously distributed. Unfortunately the use of a CDHMM needs a lot of processing power and is therefore not well suited for systems with limited processing power or even with integer arithmetic only. In these cases a discrete density hidden Markov model (DDHMM) is used. The observation density can now only take on discrete values. Hence we need discrete-valued features. One way to generate such discrete-valued features is by using a vector quantization (VQ). A VQ with a full search (all vectors are compared with all codebook vectors) however needs also a lot of processing power. Hence we need an efficient approach in order to use the VQ and therefore the DDHMMs efficiently.

This semester thesis describes an algorithm to generate a hierarchical codebook which can be used to make a hierarchical search instead of a full search. There exist a lot of algorithm to generate a hierarchical codebook. Special to the algorithm described here is however, that we start from an already existing full codebook and then decimate this codebook to get a tree-like, i.e. a hierarchical codebook in the end. To do this we sequentially delete the codebook vectors which lead to the least increase of distortion until the required number of codebook vectors for the next higher tree level is reached. For every codebook vector on the new tree level we then compute the frequency of the daughter nodes of the level below and make a connection if the frequency value is higher than a given threshold. The decimation approach is iterated until the root node is reached.

The decimation algorithm and other test and evaluation functions were implemented in MATLAB. Also a second approach to generate the codebook vectors of every tree level was tested: instead of decimation the codebook vectors were created using the LBG algorithm. In the end the two methods to generate a hierarchical codebook were compared to a codebook with a full search. As it turns out a new vector can now be quantized much more efficiently. The vector has to be compared to a smaller number of codebook vectors if the tree-like codebook is used. This makes the whole quantization algorithm a lot faster. However using a hierarchical codebook, especially with higher values for the threshold used to determine the daughter nodes, leads to some errors which results in bigger distortion values. An equality between a quantization with a full and a hierarchical codebook is also nearly impossible.

What could be next steps? The algorithm should be implemented in a programming language other than MATLAB (no efficient matrix or vector multiplication) to get real results regarding required processing time and power. Also an application of the algorithm to a speech recognition task with a lot of training and test data would be required to get consistent results.

1 Specifications

This section describes the given specifications. The original task description which contains further information can be found in appendix B.

1.1 Problem Description

Statistical speech recognition today is mainly based on hidden Markov models (HMM) [1, chapter 5] more precisely on continuous density hidden Markov models (CDHMM) [1, chapter 5.5]. CDHMMs have continuously distributed observation densities. Although CDHMMs lead to a good speech recognition rate they are not well suited for systems with limited processing power or integer arithmetic only. Therefore discrete density hidden Markov models (DDHMM) [1, chapter 5.4] are used. Their observation distribution can only take discrete values. In a first step the used features have therefore to be quantized to discrete values. For this often a vector quantization (VQ) [1, chapter 4.7] is used. This leads to an other problem. A vector quantization with a full codebook compares every test vector to every codebook vector and therefore also uses a lot of processing power. In order to use a DDHMM on systems with limited processing power the vector quantization has to be made less demanding in the first place. In this semester thesis an approach to generate a hierarchical codebook is presented. With a hierarchical codebook every test vector has to be compared only to a part of the codebook vectors and therefore the whole vector quantization runs faster and more efficiently.

1.2 The Decimation Algorithm

The algorithm can be described as a hierarchical codebook generation by decimation and works as follows:

1. We start with a full codebook (the initial codebook) and a sufficiently large number of training vectors. The initial codebook can be generated using any algorithm (e.g. LBG [1, p. 104]). These initial codebook vectors (named S_0) form the lowest level (level 0) of the tree, i.e. the leave nodes. The number of leaves nodes is described as M_0.

2. In a next step we move to the next higher tree level $l \rightarrow l + 1$.

3. This higher level l only contains $M_l = M_{l-1}/B$ nodes, where B denotes the decimation factor. Therefore in every step we decimate the number of nodes by a factor B and form the new set S_l of nodes. To get these M_l nodes we sequentially delete the nodes with the least increase of distortion. To calculate the distortion values we use the given set of training vectors.

4. Now we have reached a set of S_l nodes of level l but we also have to connect these nodes with the lower level in order to get a tree. We therefore search for each node of level l a number of daughter nodes of level $l - 1$ with frequency values higher than a threshold f_t. In order to compute the frequency values we once again use the training vectors. The frequency threshold f_t is fixed at the beginning of the algorithm.

5. We now check if the number of nodes of level l (M_l) is greater than the decimation factor B. If that is the case we repeat the algorithm from step 2. Otherwise we can add the root node on top of the generated tree and connect all nodes of level l to the root node.

Figure 1 summarizes the algorithm in a flow chart. As we can see from this figure, deleting the nodes of a level and finding the daughter nodes can be done separately. However in a actual implementation this may not be the ideal solution. For an example tree with decimation factor $B = 3$ see figure 2.

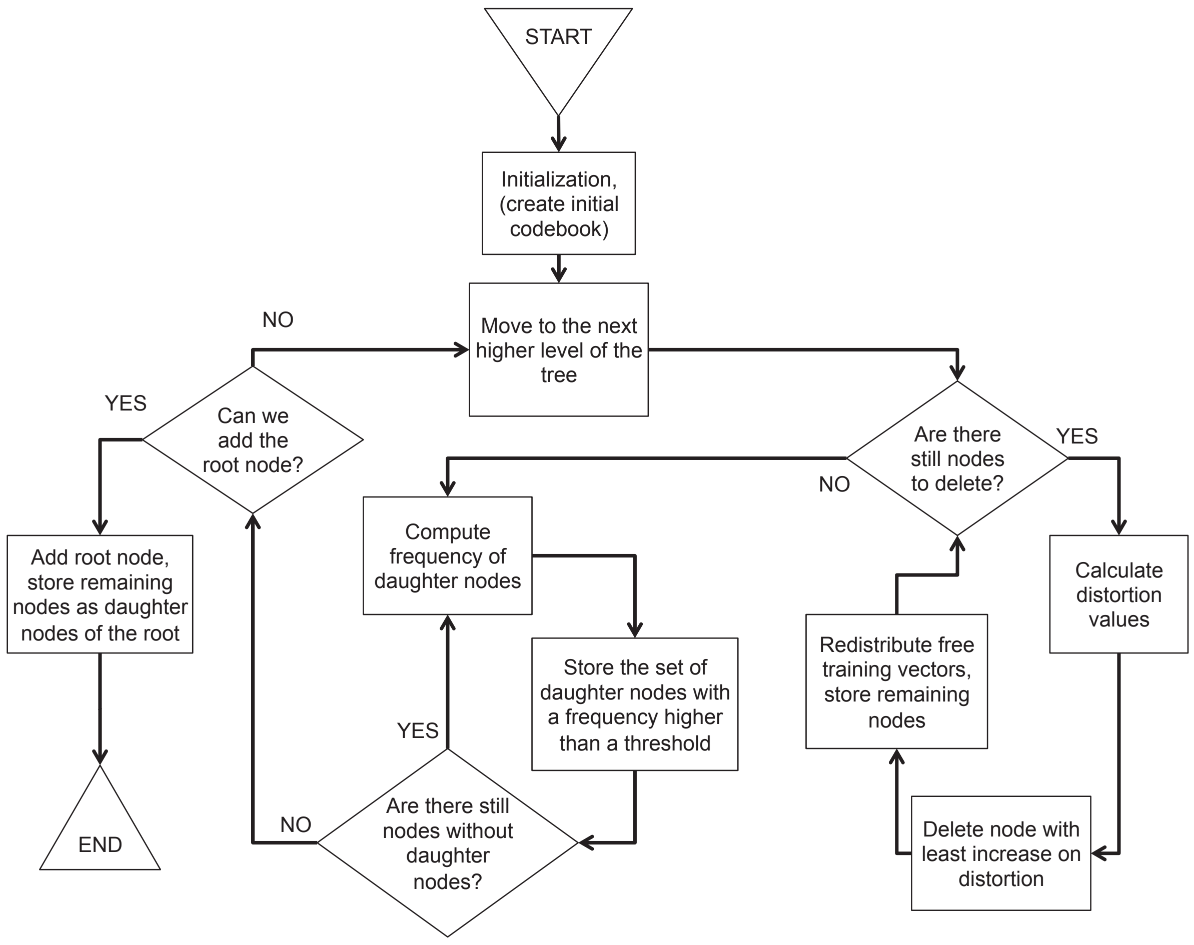
START
Initialization, (create initial codebook)
Move to the next higher level of the tree
NO
YES
Can we add the root node?
Are there still nodes to delete?
YES
NO
Compute frequency of daughter nodes
Add root node, store remaining nodes as daughter nodes of the root
Store the set of daughter nodes with a frequency higher than a threshold
Redistribute free training vectors, store remaining nodes
Calculate distortion values
YES
NO
Are there still nodes without daughter nodes?
Delete node with least increase on distortion
END

Figure 1: The decimation algorithm represented in a flow chart.

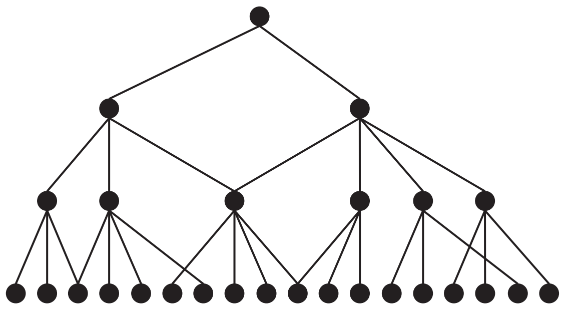

Figure 2: An example of a hierarchical codebook of initial size 18 and decimation factor $B = 3$ created with the decimation algorithm.

1.3 Goal

The goal of this semester thesis is to implement and test the decimation algorithm. Does this approach to generate a hierarchical codebook work? How does the performance of the algorithm (number of correctly distributed vectors, average distortion value and number of vector quantizations) change with different values for the parameters B and f_t? Can we use this algorithm to train a DDHMM which then can be used in a speech recognition task?

2 Preconditions

2.1 Distortion and Distance Measurement

To describe the distortion, the error created through the vector quantization, a distance measurement is needed which satisfies the following conditions [1, p. 101]:

$$0 < d(\mathbf{x}, \mathbf{y}) < \infty, \qquad \forall \mathbf{x} \neq \mathbf{y} \tag{1}$$

$$d(\mathbf{x}, \mathbf{x}) = 0 \tag{2}$$

A distance measurement which is often used and satisfies these conditions is the squared Euclidean distance. Throughout this semester thesis the squared Euclidean distance is used and is defined as follows:

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||^2 = \sum_{i=1}^{D} (x_i - y_i)^2 \tag{3}$$

for two vectors $\mathbf{x}$ and $\mathbf{y}$ of dimension D. We can therefore say a training vector has the smallest distortion value to a codebook vector or we can equally say a training vector has the smallest distance to a codebook vector. The squared Euclidean distance is also used in a lot of real speech processing tasks and is therefore well suited as a distance measurement.

2.2 Random Generated Data

To test and evaluate the created functions a lot of test data were needed. The number of available real speech data is however limited and not always well suited to test specific parts of an application. Therefore random training and test vectors were used. This leads to the question how to generate these data such that they are similar to real speech data and the results can be compared. As it turns out multivariate Gaussian distributions [1, p. 113] can model real speech data quite well. For example they are used to describe the observation densities of CDHMM. Further information about the generation of the random test and training data can be found in section 4.1.

2.3 Number of Initial Codebook Vectors

To generate the initial codebook vectors (see section 1.2 item 1) the LBG algorithm is used. The LBG algorithm normally generates only codebooks with a size of a power of two. In every iteration every codebook vector is split into two new vectors and these vectors are once again ”optimally” distributed. It is therefore not clear how to generate a codebook vector of arbitrary size using the LBG algorithm. Which vectors do we split if the size of the requested codebook vectors is not a power of two? Therefore the number of initial codebook vectors was restricted to a power of two. This is often done in real applications (e.g. the number of different observations in a DDHMM) and therefore not a big restriction. It is however important to note that the algorithm with decimation would work with an initial codebook of arbitrary size.

3 Solution Process

In a first step some literature search was done in order to compare the given algorithm (section 1.2) with other algorithms which can be used to generate a tree-like codebook. As it turns out, the idea to generate a tree structured search graph is absolutely not new. There are a lot of other approaches. In the literature they are often called tree-structured vector quantizations. Some algorithms start from one codebook vector (e.g. the centroid of all training vectors [2, p. 485]) and build the tree from the root node to the leaves. In most cases only a binary tree is generated [2, p. 485], [3, p. 568], [4, p. 400]. The algorithm presented in this semester thesis however starts from the leave nodes, i.e. an already given and optimized codebook and then builds the tree up to the root. Every node can have an arbitrary number of daughter nodes.

In a next step a simple version of the decimation algorithm was implemented which recomputed in every step all the required information and did not take advantage of previous calculated values. Therefore the program needed a lot of processing time and only examples with a small number of data (training and codebook vectors) could be used to test the program in a reasonable time. This had the advantage that part of the calculated trees and, as well, the whole algorithm could be easily checked. Beside the actual algorithm to generate the hierarchical codebook also scripts and functions to generate the random training and test data and to quantize new data using the tree-like codebook were programmed. Some basic functions (LBG algorithm for the initial codebook and a function for a full vector quantization) were provided by the institute TIK and only some minor changes were needed.

In order to check the created programs, a test function, which only uses the already given (and assumed to be correct) vector quantization function to generate the tree, was implemented. Unfortunately the runtime of this program was quite bad and so only examples with a small number of test and training vectors could be checked. However the created functions seemed to be correct. After verifying the correctness of the implemented programs a new version of the tree generating algorithm was implemented. This time a lot of the computed data was temporarily stored for the next steps and only the parts which changed from one to an other step were recalculated (see section 4.2 part 1). This program runs much faster and therefore bigger examples can be computed.

Then a different way of implementing the tree generating algorithm was created. Instead of deleting the nodes in every level by decimation, the LBG algorithm was used to create the required number of nodes of the higher level. A node on level l is now completely independent of the nodes on level $l - 1$ and $l + 1$. But perhaps it is better to distribute the codebook vectors "optimally" (using the LBG algorithm) than keeping the old ones. With this the number of codebook vectors increases, therefore also slight changes to the test programs were necessary.

Now big examples with millions of training and test vectors were computed and analyzed. The results of these simulations can be found in section 5. In the end a hierarchical codebook was created from real speech data and used to train DDHMMs. These were then used in a speech recognition task.

4 Program Description

This section presents a description of the developped functions and scripts. The order of the subsections represents the flow of the whole program.

4.1 Training, Test and Codebook Vector Generation

To test the programs excessively random training vectors were created (see section 5 for the results and tests with real speech data). There are three different ways to generate the training and test vectors:

1. The vectors are uniformly distributed over a specific area. Although this distribution is perhaps not the most natural one, the implementation is quite easy. The MATLAB function $rand$ was used to generate the required number of vectors. For different experiments, different seed values for the rng function were used.
2. Mean vectors are uniformly distributed (using the $rand$ function) and for every mean vector a specific number of training and test vectors is generated according to a normal Gaussian distribution (using the $randn$ function). This can be seen as a fixed number of training and test vectors for every codebook vector (represented by the mean vectors).
3. Mean vectors are uniformly distributed (using the $rand$ function) and for every mean vector a fixed number of training and test vectors is generated according to a multivariate Gaussian distribution (using the $mvnrnd$ function). This distribution was used for the numerous simulations in the evaluation section.

The next step after the training and test vector generation was the creation of the initial codebook vectors in the hierarchical codebook tree. For the given random training vectors a predefined number of codebook vectors was generated using the LBG algorithm. For this an already existing function was used. Especially for larger examples the generation of the codebook vectors with the LBG algorithm could take some time. Therefore the generated training and codebook vectors were saved in a file. In the same way hierarchical codebooks with the same training and initial codebook vectors but different B and f_t parameters could be generated without recalculating the initial codebook every time.

4.2 Hierarchical Codebook Generation with Decimation

This function represents the main part of the whole program; the hierarchical codebook generation. After some initialization steps (determine the maximum level of the tree etc.) there are two main parts: 1. The determination of the codebook vectors of level l by decimation and 2. the process of finding daughter nodes for every codebook vector.

Part 1: The decimation approach deletes the nodes with the least increase of distortion. Therefore, in a first step, for every codebook vector the possible distortion increase is calculated should this codebook vector be deleted. How can we find these values? Remembering the calculation of the Euclidean distance (3) every training vector $\mathbf{x}$ adds $d(\mathbf{x}, \mathbf{y_2}) - d(\mathbf{x}, \mathbf{y_1})$ to the distortion increase of codebook vector $\mathbf{y_1}$ where $\mathbf{y_1}$ is the codebook vector nearest to $\mathbf{x}$ and $\mathbf{y_2}$ the codebook vector second nearest to $\mathbf{x}$. To calculate these values in MATLAB efficiently we can use the following "trick": For every training vector $\mathbf{x}$ we first create a matrix $\mathbf{A}$ which contains as many copies of $\mathbf{x}$ as there are codebook vectors. We then build the difference $\mathbf{L} = \mathbf{A} - \mathbf{B}$ where $\mathbf{B}$ is a matrix which contains every codebook vector one time (all vectors are row vectors). Now we multiply every entry of $\mathbf{L}$ with its own (in MATLAB notation: $\mathbf{L}.{*}\mathbf{L}$) and sum all values in direction of the row of the resulting matrix. Now we just have to find the minimum value of the sum to get $d(\mathbf{x}, \mathbf{y_1})$ and the second smallest value to get $d(\mathbf{x}, \mathbf{y_2})$. In MATLAB this is faster and more efficient than using a loop over all codebook vectors. We can do this for every training vector and then delete the codebook vector with the least increase of distortion. All his training vectors are redistributed to the remaining codebook vectors. This leads to a change of the distortion values for the remaining codebook vectors. Therefore their values have to be recomputed in order to determine the next codebook vector, to be deleted. A simple approach to determine the new distortion values is to recalculate all values for all remaining codebook vectors. Although for this approach nearly no values have to be stored, the runtime behavior is quite bad. Therefore in a second implementation only the distortion values of training vectors which lead to a change are recalculated. Those are all the training vectors which are nearest to the deleted codebook vector, so to get these the distribution training vector to codebook vector has to be stored. Also the training vectors which are second nearest to the deleted codebook vector will now result in a different distortion value. So also the second nearest codebook vector for every training vector has to be stored (see figure 3 for a visual representation). We therefore have to save two additional values for every training vector. For an example with a lot of training data this additional storage space requirement could be problematic. Using this approach the algorithm to generate the hierarchical codebook runs much faster.

Part 2: We now have to connect the two levels in order to get a hierarchical codebook. We can also say we are looking for the daughter nodes of the codebook vectors of level l. For every codebook vector of level l (CBV_l) we do the following: First of all we count the total number of training vectors which are nearest to CBV_l ($= n_l$). This can be easily done because we have stored the distribution training vector to codebook vector. Secondly we determine for every codebook vector of level $l-1$ (CBV_{l-1}) how many of the training vectors which were nearest to it are now nearest to CBV_l. For this a vector quantization of all training vectors with the codebook vectors of level $l-1$ is needed. Should this number related to n_l (the frequency) exceed a predefined value f_t a connection from CBV_l to CBV_{l-1} is made. The codebook vector of level $l-1$ is now a daughter node of the codebook vector of level l.

As long as the number of remaining codebook vectors is bigger than B, the decimation factor, we repeat this two steps. Afterwards we add the root node and add all remaining codebook vectors as daughter nodes to the root.

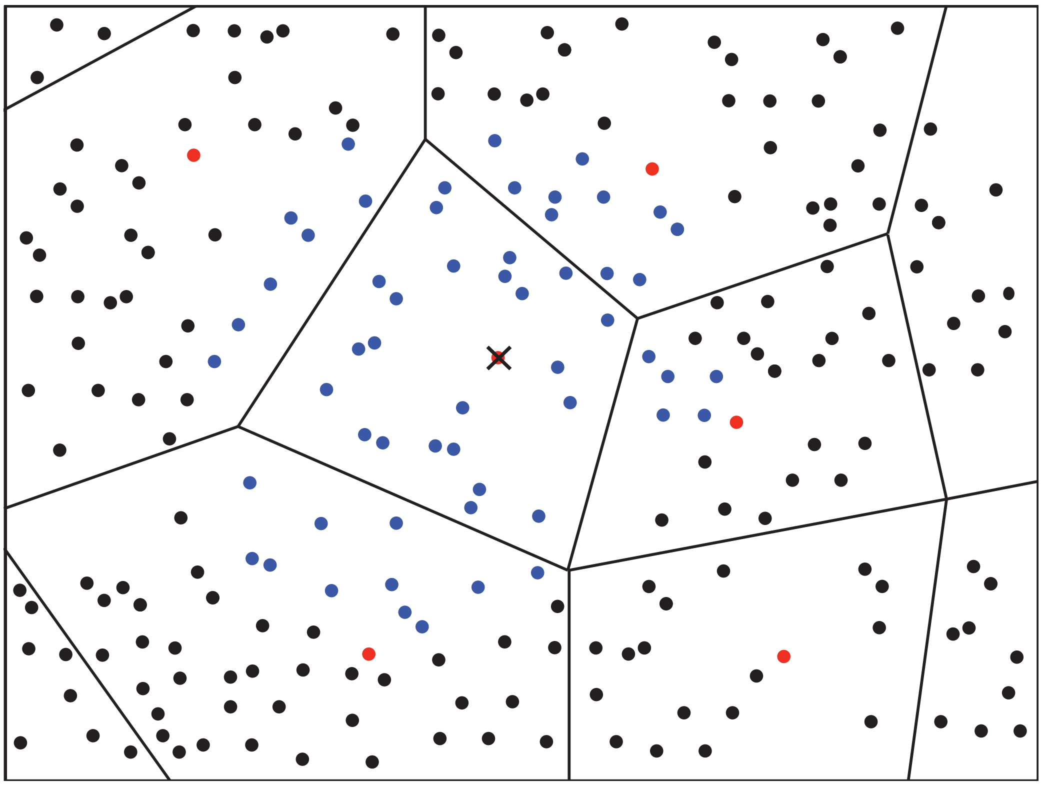

Figure 3: Red dots represent codebook vectors, blue ones training vectors which lead to additional distortion values and black dots are training vectors which lead to no additional distortion values. The codebook vector in the middle will be deleted. So only for the blue training vectors the distortion values have to be recalculated!

4.3 Hierarchical Codebook Generation with LBG

An other idea was to generate the new codebook on a next higher level with the LBG algorithm instead of the decimation approach. The ”optimal” distribution of the LBG algorithm could perform better than the decimation algorithm. For this implementation some changes to the code were needed. Part 1 of section 4.2 now uses an already given function to calculate the LBG algorithm to generate the needed number of codebook vectors. Therefore in every level new codebook vectors are generated, which were most likely not present in the tree levels below. This is especially important for the test functions (see section 4.4). Part 2 works as in section 4.2.

The LBG algorithm normally generates only codebooks of size 2^i. In order to compare a hierarchical codebook generated with the LBG approach to one generated with the decimation approach, only decimation factors (B) of size 2,4,8,... are possible.

4.4 Test and Quantization Programs

In order to test the created hierarchical codebook, a vector quantization function was needed which could handle the hierarchical codebook. The quantization starts with the daughter nodes of the root node. These codebook vectors are the same for all vectors which are going to be quantized. But after this step for every vector the correct daughter nodes have to be used. The function therefore saves the intermediate nearest codebook vector for all vectors and then compares all vectors which are nearest to the same codebook vector to his daughter nodes. This is iterated until all vectors reach the bottom of the tree. Compared to a vector quantization with a full search, every vector has to be compared to less codebook vectors but this process is repeated multiple times for every level of the tree and some intermediate state has to be saved. If the hierarchical codebook was generated with the LBG algorithm some adaptions were needed because the codebook vectors of level $l > 0$ are not present in the initial codebook vector set.

Also a function to compare the hierarchical codebook to a full one was created. It uses the vector quantization from above but also quantizes every vector with the full codebook (the lowest level of the tree). The function outputs: the time used to quantize all the vectors for both variants (full and tree), the distribution for every vector quantized with the full codebook and with the hierarchical one, the number of vectors which are distributed to the same codebook vector, the average distortion value which results from the quantization and the average comparisons vectors to codebook vectors which were needed to quantize the test vectors. These values are used in the evaluation (see section 5).

5 Evaluation

In this section the main results are presented. For all computations with random values a multivariate Gaussian distribution (see section 4.1 item 3) was used. Information about the computer specifications and the parameters to reproduce the results can be found in appendix A. In the first subsection some general results regarding the hierarchical codebook are presented.

5.1 General Results

5.1.1 Equality of a Vector Quantization with a Full and a Hierarchical Codebook

It is quite obvious that for bigger values of f_t (the frequency threshold for the daughter node determination) a vector quantization with a hierarchical codebook will lead to different results than a quantization with the full codebook. Some connections in the tree are omitted and a vector can not go back to a different branch of the tree once a decision is made. But is that also the case if f_t is set to zero? That means every codebook vector of level l of the tree will make a connection to every daughter node of level $l-1$ as soon as at least one training vector nearest to the codebook vector of level $l-1$ is now nearest to the codebook vector of level l. Will the hierarchical codebook in this case quantize every vector to the same value as a quantization with the full, i.e. normal codebook? The answer is no! The problem is illustrated in figure 4 and 5. It is important to note that the number of training and codebook vectors used for this example are by no means comparable with a real application. This example just illustrates the problem.

In figure 4 the training vectors (green plus signs) and the initial codebook vectors (red crosses) are plotted. The blue vector (not part of the training vectors) should now be quantized. If we go through the created hierarchical codebook (generated using the decimation algorithm with a decimation factor of 2 and a frequency threshold $f_t = 0$) drawn in figure 5 we first have to decide between codebook vector 4 and 6. Obviously codebook vector 6 is nearer to the blue vector. We then look at the daughter nodes of codebook vector 6: vectors 1, 3 and 6. Codebook vector 1 is nearest to the blue vector. Codebook vector 1 has only itself as daughter node. So the blue vector will be quantized as 1. Would we now quantize the blue vector with the full codebook, it would be quantized as 5. As we can see even though f_t is zero the blue vector is quantized to different values using the hierarchical codebook and the full codebook. Why did that happen? In the first round of the decimation algorithm codebook vectors 2,5,7 and 8 are deleted. Then their training vectors have to be distributed to the remaining codebook vectors. Unfortunately all training vectors of codebook vector 5 are now distributed to codebook vector 3 none is distributed to codebook vector 1. Therefore codebook vector 5 is not a daughter node of codebook vector 1. There is no connection in the tree. If there had been more training vectors (especially in the region of the blue vector) this problem would not have happened. This problem can however never be eliminated completely as can be also seen in the examples with a lot of random training vectors in the following subsections. It is important to note that all the training vectors would indeed be quantized to the same values as with a full codebook but this is of course not a practical application, we want to quantize new vectors. The approach used to generate the hierarchical codebook (decimation or LBG) does not matter.

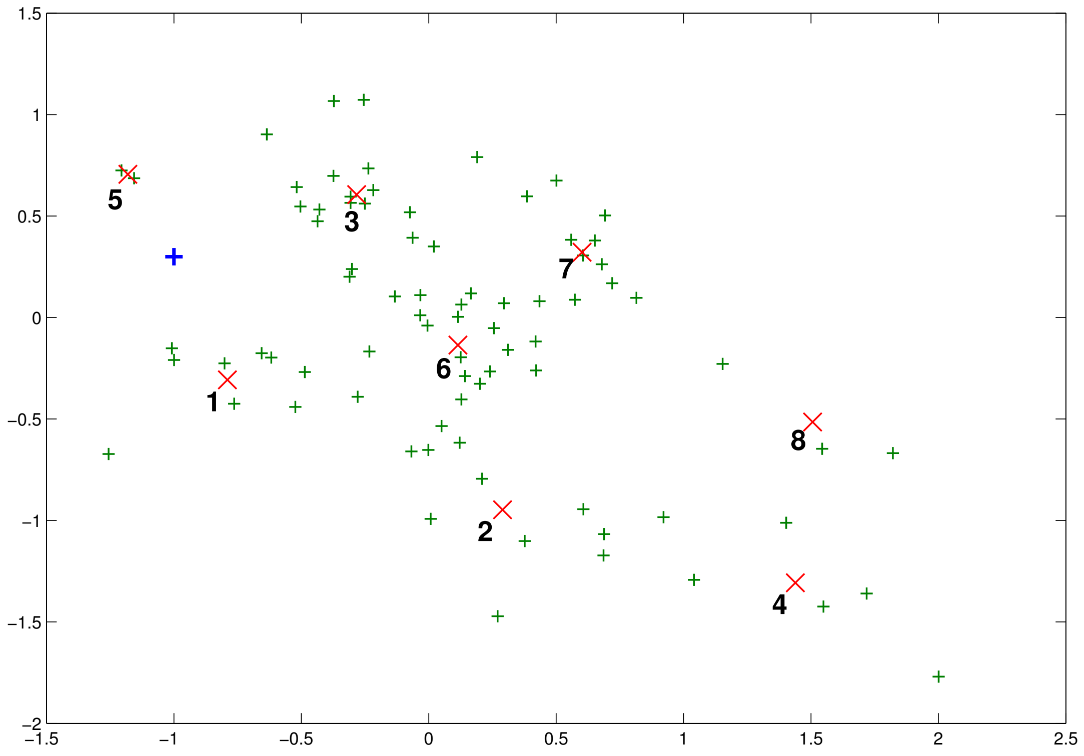

Figure 4: Training vectors (green plus signs), codebook vectors (red cross signs) and one test vector (blue plus sign) in a small example to illustrate the inequality of a full codebook and a hierarchical codebook.

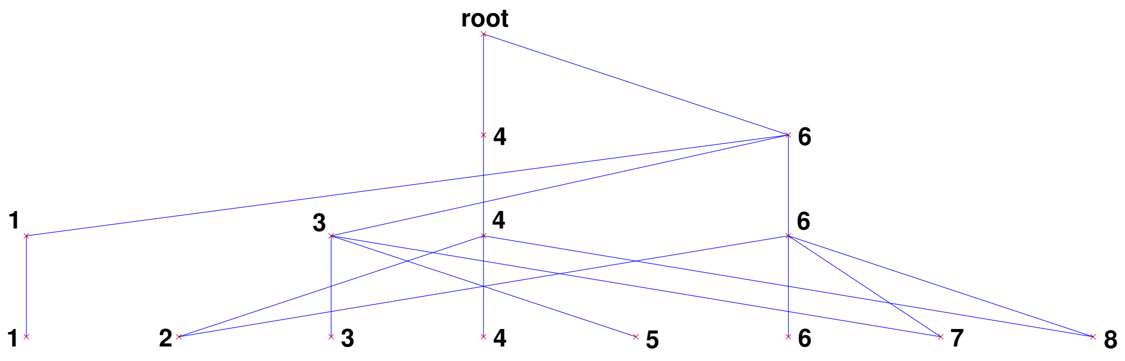

Figure 5: A hierarchical codebook generated from the training and codebook vectors of figure 4 using the decimation approach with parameters $B = 2$ and $f_t = 0$.

5.1.2 Degenerated Trees

There are two cases in which special or incomplete trees can be generated.

Case 1: Figure 6 shows an example of the first case. A codebook vector of level $l-1$ which is not present in level l (got deleted by the decimation step from level $l-1$ to level l) has no parent node in level l. This is a consequence of the used algorithm. All the training vectors nearest to the codebook of level $l-1$ are distributed to the codebook vectors of level l. Unfortunately no codebook vector of level l gets enough training vectors so that the frequency is higher than the threshold f_t. Therefore no connection is established. For higher values of f_t this can of course appear more frequently. This case leads to fewer correctly quantized vectors (some initial codebook vectors can never be reached) but does not lead to a general problem with the quantization algorithm. One way to solve this problem would be to connect a codebook vector of level $l-1$ which has no parent node to the nearest codebook vector of level l. It is important to note that this possible solution was not applied to any of the simulations in the following subsections. This problem can also appear in a hierarchical tree generated with the LBG algorithm.

Case 2: Figure 7 shows an example of the second case. This case is very rare but also more common for higher values of f_t. A codebook vector of level l ($l > 0$) has no daughter node in level $l-1$. This is a severe problem because the codebook tree is now incomplete. A vector can be quantized up to the codebook vector in level l but has then no possible daughter nodes to be further quantized. This problem can occur if the number of the training vectors nearest to the codebook vector in level $l-1$ is so small, compared to the training vectors nearest in level l, that not even the codebook vector itself has a frequency value higher than the threshold f_t. At the same time the problem from case 1 occurs. This problem is severe and has to be solved in order to quantize arbitrary vectors with the hierarchical codebook. The simplest solution is to take itself always as a daughter node no matter what the frequency value is. This solution is applied in the simulations below but the problem did only appear once (marked in appendix A). This problem could possible also appear in a hierarchical codebook generated with the LBG algorithm. However it was never observed so far. In the LBG case codebook vectors of level l are independent of codebook vectors of level $l+1$ and $l-1$ a possible solution is therefore not so obvious.

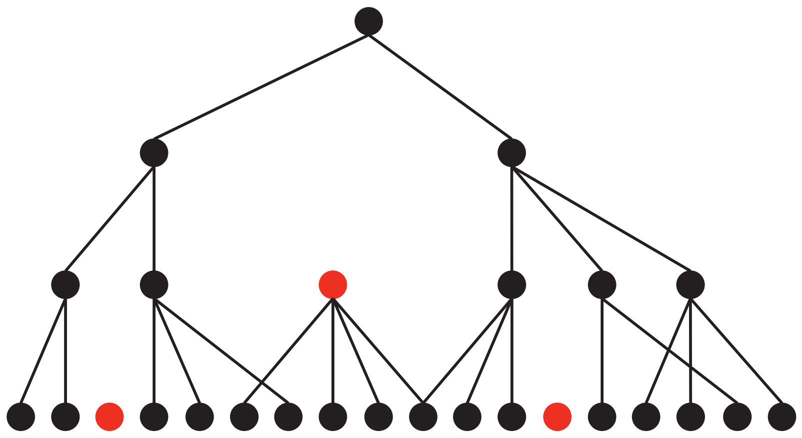

Figure 6: The red nodes are examples for the case 1 degeneration of the tree.

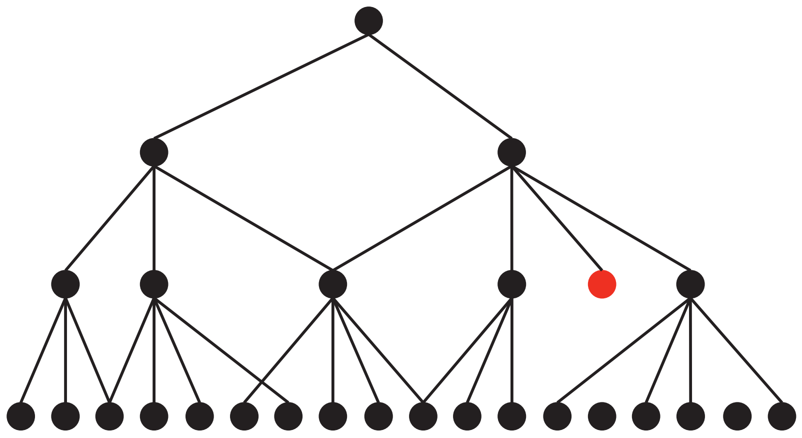

Figure 7: The red node is an example for the case 2 degeneration of the tree.

5.2 Evaluation for Different Values of Parameter f_t

In this section the quantization performance of a hierarchical codebook generated with the decimation or the LBG algorithm is analyzed for different values of the frequency threshold f_t. For every simulation 2'048'000 training vectors of dimension 12 were used to create a hierarchical codebook with 2'048 initial codebook vectors and a decimation factor $B = 4$. 2'048'000 different test vectors (generated with the same multivariate Gaussian distribution as the training vectors) were then quantized with the full and the hierarchical codebook. Every simulation was repeated 8 times with different values for the random seed. Consult appendix A for all the necessary values to repeat the simulations. For all the plots in this and the next section the MATLAB function boxplot was used. The red line inside the box represents the median, the edges of the box correspond to the 25th and 75th percentiles. The whiskers extend to the most extreme data points, which are not considered as outliners. Outliners are separately marked with red plus signs [5].

In figure 8 the percentage of equally quantized test vectors are plotted for different values of f_t. On the left side the hierarchical codebooks were generated with the decimation algorithm and on the right side with the LBG algorithm. In general we can see that the number of correctly quantized vectors decreases with higher values of f_f. For f_t values 0, 0.005 and 0.01 (zero, a half or one percent) on average more than 97% of all test vectors are quantized correctly. Also the variance between the different simulations is very low. There are no obvious differences between the decimation and the LBG approach. For f_t values 0.05 and 0.1 the variance between the different simulations increases and the LBG algorithm performs better. The "optimal" distribution of the LBG algorithm on every level seems to lead to more daughter nodes and therefore also more correctly distributed test vectors. Even with $f_t = 0$ there was no example for which all the test vectors were correctly quantized (see section 5.1.1).

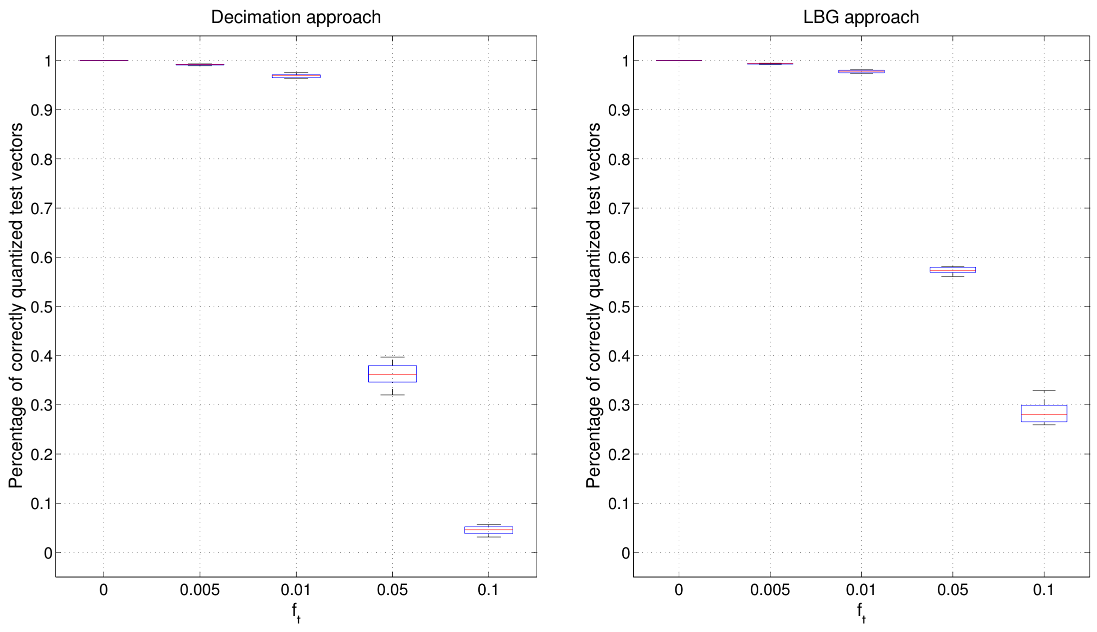

Figure 8: Percentage of correctly quantized test vectors for different f_t values. The hierarchical codebook was tested with 2'048'000 test vectors.

In figure 9 the relative distortion increase is plotted for different values of f_t. On the left side the hierarchical codebooks were generated with the decimation algorithm and on the right side with the LBG algorithm. Once again for f_t values 0, 0.005 and 0.01 there are nearly no differences between the two approaches and the different simulations (low variance). The distortion values are also comparable to a codebook with a full search. For f_t values of 0.05 and 0.1 the distortion values are higher and the LBG algorithm performs better. These observations correlate with figure 8: more correctly quantized test vectors lead to smaller average distortion values. This correlation also shows that wrongly quantized test vectors are not quantized to an arbitrary value. They are still near the correct codebook vector, i.e. the correct quantization value.

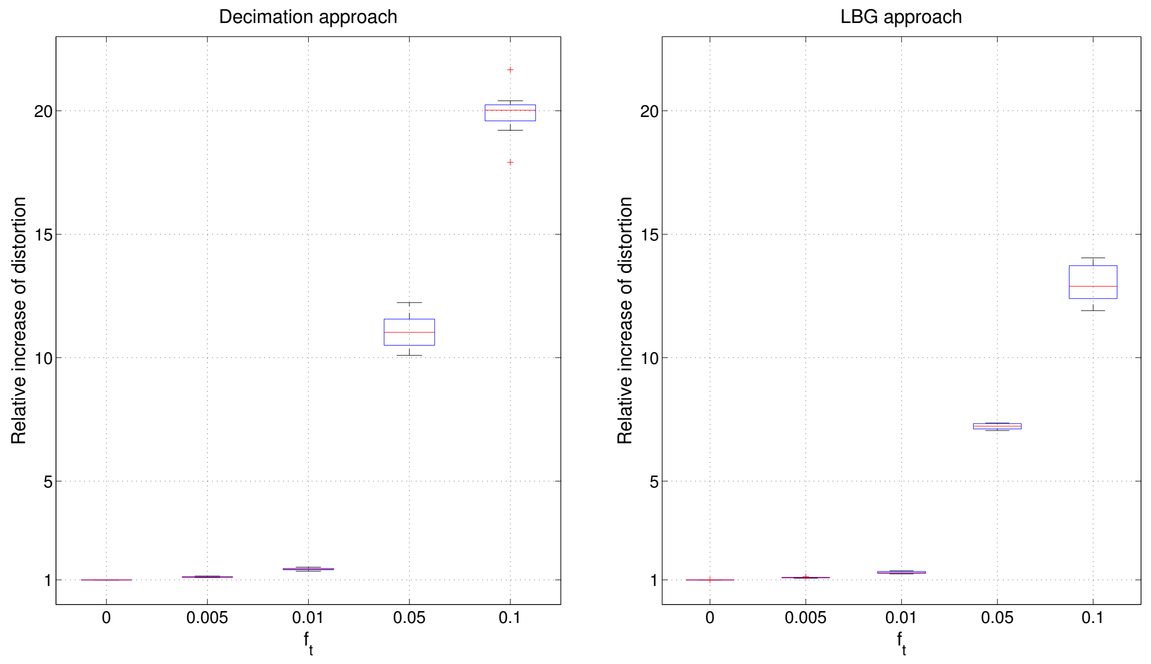

Figure 9: Relative increase of distortion for different f_t values. The total distortion values are divided by the distortion values of a full codebook search.

An implementation in MATLAB is not comparable to an implementation in a programming language which does not support fast matrix multiplications. The time used to quantize the test vectors is therefore not an appropriate measurement for the efficiency of the hierarchical vector quantization. Better suited is the following value: for every test vector the number of comparisons with codebook vectors is measured. In a full codebook every test vector has to be compared with every codebook vector. In a hierarchical codebook every test vector has to be compared with only a part of the codebook vectors. In figure 10 the average number of comparisons for a test vector is plotted for different values of f_t. On the left side the hierarchical codebooks were generated with the decimation algorithm and on the right side with the LBG algorithm. With higher values of f_t the number of comparisons decreases. Interesting is the number of comparison for $f_t = 0$: Even though we take all possible nodes as daughter nodes the number of comparisons is already quite low. Compared with a full search (2'048 comparisons for this example) more than a factor of 20 fewer comparisons are needed. For f_t values 0, 0.005 and 0.01 the LBG algorithm needs fewer comparisons. If we compare this to the results from figure 8 and 9 we can conclude that the LBG approach produces nearly the same performance values but achieves this with fewer comparisons! For $f_t = 0.05$ both algorithms need nearly the same number of comparisons and for $f_t = 0.1$ the decimation algorithm needs fewer comparisons. However that does not mean, that the decimation algorithm performs better since it also leads to a lot less correctly quantized test vectors (see figure 8).

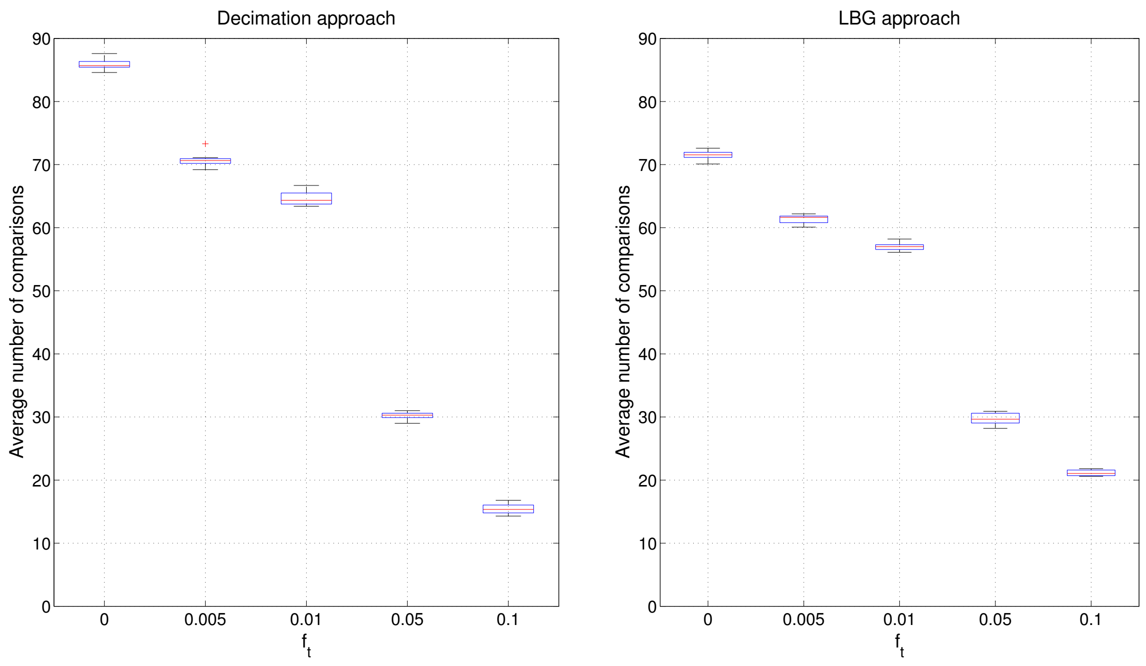

Figure 10: Average number of comparisons (test vector to codebook vectors) for different f_t values. A quantization with a full codebook needs 2'048 comparisons for one test vector.

In general we can conclude: for small f_t values 0, 0.005 and 0.01 both approaches to generate the hierarchical codebook lead to nearly the same number of correctly quantized vectors, the LBG approach however needs less comparisons. Compared to a quantization with a full codebook the hierarchical codebook performs quite well up to $f_t = 0.01$ but needs a lot less comparisons.

5.3 Evaluation for Different Values of Parameter B

In this section the value for the parameter f_t is fixed at 0.01 but the decimation factor B changes. All the values of B are a power of two. As in section 5.2 2'048'00 training vectors of dimension 12 were used to generated the hierarchical codebooks. Then 2'048'000 different test vectors were quantized and the results with the hierarchical codebook were compared to a quantization with a full codebook. Every simulation was repeated 8 times with different values for the random seed. The same random seed values as in section 5.2 were used. The results for $B = 4$

therefore correspond to the values with $f_t = 0.01$ in section 5.2. Consult appendix A for all the necessary values to repeat the simulations.

In figure 11 the percentage of equally quantized test vectors are plotted for different values of B. On the left side the hierarchical codebooks were generated with the decimation algorithm and on the right side with the LBG algorithm. For B values 2,4 and 8 the number of correctly quantized test vectors behaves more or less as expected. The higher the B value the less tree levels are generated and therefore less possibilities to distribute the test vectors exist. For B values 16 and 32 the behavior seems strange. Why quantizes the hierarchical codebook with $B = 32$ more vectors correctly? To understand this we first note that the hierarchical codebook for $B = 16$ and for $B = 32$ in both cases consists of 4 tree levels (including level zero and the root node). For $B = 16$ the number of nodes per level is 2'048, 128, 8 and 1. For $B = 32$ the number of nodes per level is 2'048, 64, 2 and 1. The number of nodes on level $l - 1$ for every node on level l is higher for the case $B = 32$. f_t is equal to 0.01 for all simulations so a lot of daughter nodes are possible. Therefore more daughter nodes for every codebook vector can be created in the case $B = 32$ and hence a test vector can be better distributed. If we compare these results to figure 13 we see indeed that the number of comparisons is higher for the case $B = 32$.

For $B = 64$ we see a huge variance in the percentage of correctly quantized vectors for the decimation approach. This is not the case for the LBG approach. For $B = 64$ only a total of 3 tree levels are generated. With the decimation approach the $2048/64 = 32$ surviving nodes of the first and only decimation round will often not be "uniformly" distributed over the whole possible vector space. A vector quantization can then be difficult. The LBG approach tries to distribute the codebook vectors optimally and will therefore lead to a more even distribution of the 32 codebook vectors for every simulation. This leads to a much smaller variance.

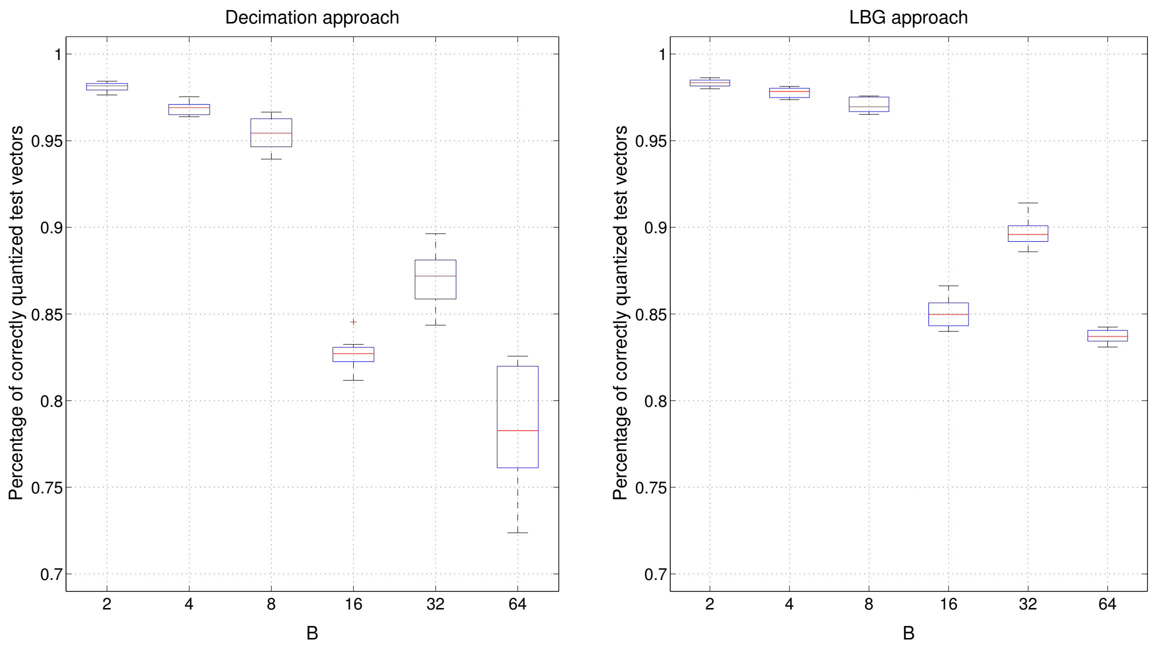

Figure 11: Percentage of correctly quantized test vectors for different B values. The hierarchical codebook was tested with 2'048'000 test vectors.

In general we can see that the percentage of correctly quantized test vectors depends much less on B than on f_t (see section 5.2). Even for $B = 64$ more than 70% of all test vectors are correctly quantized. It is nonetheless interesting that the quantization performance does not solely depend on f_t. Also the number of tree levels and the ratio of nodes between two levels are important.

Figure 12 shows the relative distortion increase for different values of B. On the left side the hierarchical codebooks were generated with the decimation algorithm and on the right side with the LBG algorithm. If we compare figure 11 and 12 we see a tight correlation between the percentage of correctly quantized vectors and the relative distortion increase. The more correctly quantized test vectors the smaller the distortion. This behavior was also observed in figure 8 and 9. Once again we see that wrongly quantized test vectors are still near the real codebook vectors.

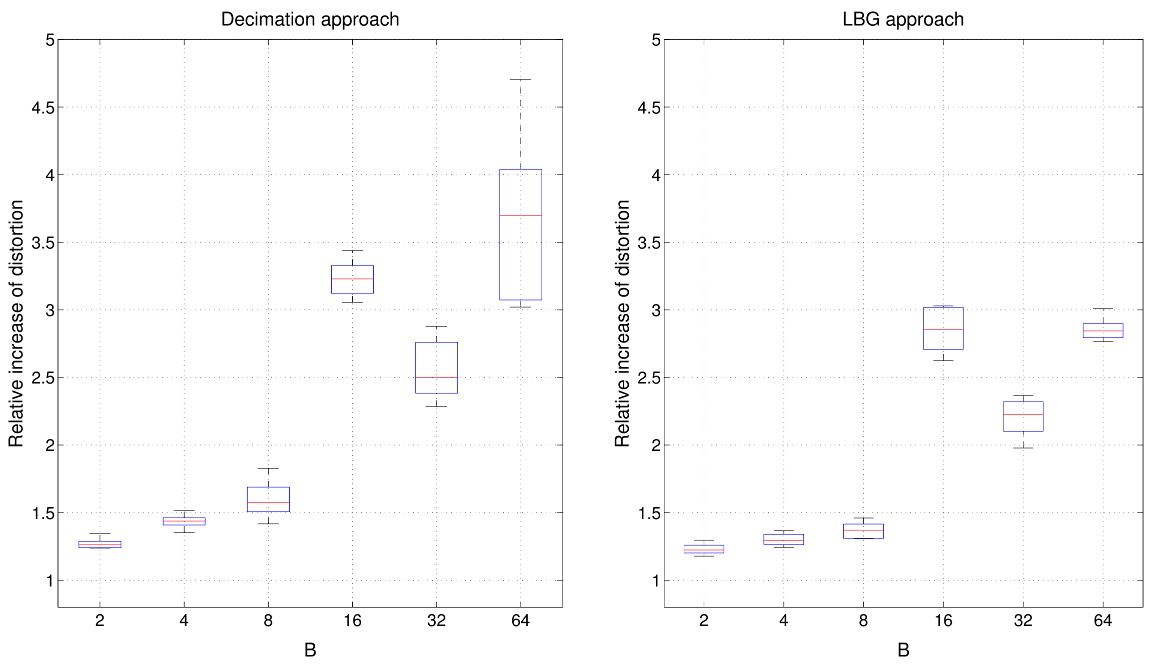

Figure 12: Relative increase of distortion values for different B values. The total distortion values are divided by the distortion values of a full codebook search.

In figure 13 the average number of comparisons test vector to codebook vectors (see section 5.2 for more information) is plotted for different values of B. On the left side the hierarchical codebooks were generated with the decimation algorithm and on the right side with the LBG algorithm. For $B = 2$ there is only a small difference between the decimation and the LBG approach. To understand this we first note that $B = 2$ will lead to a total of 12 tree levels. This is quite a high value. Between every two tree levels daughter nodes are created. Therefore even the ”optimal” distribution of the LBG algorithm will not drastically decrease the number of daughter nodes. For all the other values of B the LBG approach needs less comparisons but results in the same of more correctly quantized test vectors (compare to figure 11). This is once again explained with the more uniform distribution of the nodes on every level when using the

LBG approach. The huge difference between $B = 16$ and $B = 32$ is already explained in the analysis of figure 11. For $B = 64$ only a total of 3 tree levels are generated. The number of comparisons is nonetheless quite high. To understand this we have to remember that all the nodes on the level below the root will automatically be daughter nodes of the root. In this case every test vector will have $2'048/64 = 32$ predefined comparisons which do not depend on the path the vector takes through the tree.

Overall we can not find a clear correlation between the value of B and the number of comparisons. Small values of B will lead to a lot of tree levels and high values of B will result in a bigger number of initial comparisons (daughter nodes of the root). A value between the maximal and minimal value of B will lead to the best results.

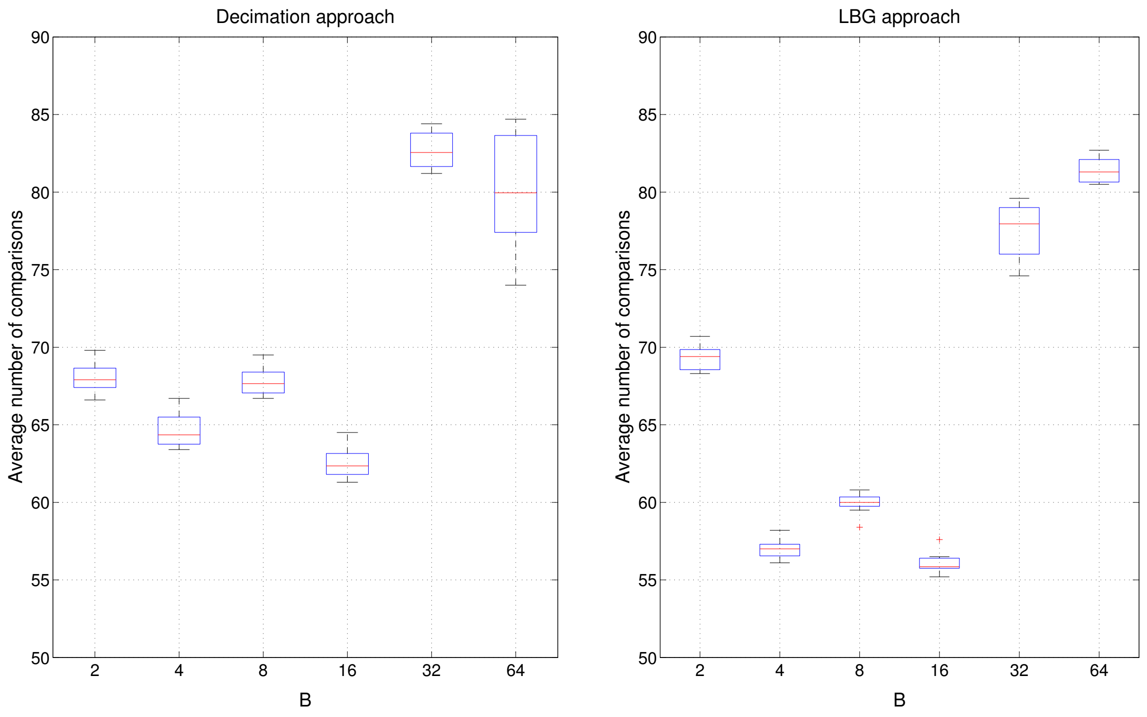

Figure 13: Average number of comparisons (test vector to codebook vectors) for different B values. A quantization with a full codebook needs 2’048 comparisons for one test vector.

5.4 Application to a Speech Recognition Task

In this section an application of the hierarchical vector quantization to a real speech recognition task is described and evaluated. The goal was to recognize german digits from zero to nine and ”Zwo” (an other pronunciation for the digit two). For this test real speech data were used. Every number was once recorded from 61 different speakers. From these data the mel frequency cepstral coefficients (MFCCs) [1, chapter 4.6] were extracted. These coefficients were then used as training vectors for a hierarchical codebook. The coefficients quantized with the hierarchical codebook where then used to train DDHMMs. One DDHMM for every digit was trained using the Baum-Welch algorithm [1, chapter 5.4] with 30 iterative rounds. To test the DDHMMs different speech data were used. 15 speakers spoke every digit 5 times. In the end the recognition rate of these data using the DDHMMs to recognize them were compared to the results using CDHMMs which were trained with the same data. As it turns out the hierarchical codebook generation works fine. The training of the DDHMMs seems also to be correct. If the training data was once again used 670 of the 671 training digits were correctly recognized. Some problems arose when the test data were used. Unfortunately a lot of digits were then recognized by none of the DDHMMs. This can happen if one of the quantized values of the MFCCs was not present in the training data. The DDHMM was then not trained for this value and it has a zero (minus infinity if the logarithm of the probabilities are used) value as an observation probability for this quantization value. Therefore the Viterbi algorithm [1, chapter 5.4], used to find the optimal state sequence, will now output a joint probability of zero for the optimal state sequence. An attempt to replace these zero values in the observation probability matrix with a small value ϵ did not really improve the recognition rate. For a hierarchical codebook with an initial codebook size of 128 and parameters $B = 4$ and $f_t = 0.1$ generated with the LBG approach the recognition rate is only 64%. Compared to a recognition rate of the CDHMMs (94.5%) this value is quite bad. On a positive note the vector quantization for the test data uses 10 times less comparisons compared to a vector quantization with a full codebook. The hierarchical codebook fulfill its purpose.

6 Conclusion

As we have seen, a vector quantization with a hierarchical codebook can quantize a vector with a lot less comparisons compared to a vector quantization with a full codebook. It runs more efficiently. An equality between a vector quantization with a full and a hierarchical codebook is however nearly impossible. But that does not mean that a wrongly quantized vector will be distributed to an arbitrary codebook vector. It is still near the correct value.

The decimation and the LBG approach start from an arbitrarily generated initial codebook (leave nodes of the tree) and grow the tree up to the root. All vectors will be quantized to one of the initial codebook vectors. This can be an advantage compared to other approaches to generate a hierarchical codebook which start from the root node and will result in unpredictable leave nodes.

For small values of f_t the decimation and the LBG approach will result in nearly the same number of correctly quantized vectors. The LBG approach can however do this with less comparisons. For higher values of f_t and B the LBG approach leads to better results. This is due to the more uniform distribution of the codebook vectors on every level when using the LBG approach. The performance of a hierarchical codebook generated with the LBG approach is therefore also less correlated to the used data. If the initial codebook is also generated with the LBG algorithm, all the needed codebook sets for the higher tree levels are already created during the initial generation. The hierarchical codebook generation is reduced to finding the daughter nodes for every tree level. The LBG approach will then need less time compared to a generation with decimation. The number of initial codebook vector and also the values of the parameter B are restricted to a power of two when using the LBG approach. For the decimation approach there are no such restrictions.

7 Recommendation for Further Projects

There is still plenty of work to do. First the quantization function for a hierarchical codebook could be implemented in a language other than MATLAB (e.g. JAVA or Python) which doesn't support fast vector and matrix multiplications. This would lead to better results regarding the used processing power and time. Then the algorithm should be applied to speech recognition tasks with a lot more test and training data to get meaningful results. One could use large collections of speech data which are available in national or international libraries. There are also some other open questions: does the correctness of the quantization of a hierarchical codebook depend on the dimension of the training and codebook vectors? Could an other approach to determine the daughter nodes lead to better results?

References

[1] B. Pfister and T. Kaufmann. *Sprachverarbeitung: Grundlagen und Methoden der Sprachsynthese und Spracherkennung*. Springer Verlag, 2008.

[2] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. *SIGGRAPH '00*, pages 479–488, 2000.

[3] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel. Speech coding based upon vector quantization. *IEEE Transactions*, pages 562–574, 1980.

[4] Ioannis Katsavounidis, C.-C. Jay Kuo, and Zhen Zhang. Fast tree-structured nearest neighbor encoding for vector quantization. *IEEE Transactions*, pages 398–404, 1996.

[5] MathWorks Inc. Boxplot documentation. `http://www.mathworks.ch/ch/help/stats/boxplot.html`, May 2014.

A Parameters Used for the Simulations

In this appendix all the values to repeat the computed simulations are given.

For section 5.2 the following values were used: For every random seed value (seed_rng): 500, 510, 520, 530, 540, 550, 560 and 570 one simulation for every f_t (ft) value: 0, 0.005, 0.01, 0.05 and 0.1 was made with the following additional parameters:

N_tv	N_cbv	N_tv_for_cbv	dim	type_rng	factor_rng	factor_rng2	epsi	B
2'048'00	2'048	1'000	12	3	500	5	0.001	4

Table 1: Used values for the simulations of section 5.2.

These simulations were made for the approach with decimation (version_HCB = 1) and for the approach with LBG (version_HCB = 3). The simulation with random seed value 500 for the hierarchical approach with $f_t = 0.1$ led to the described degenerated tree case 2 (see section 5.1.2).

For section 5.3 the following values were used: For every random seed value (seed_rng): 500, 510, 520, 530, 540, 550, 560 and 570 one simulation for every B value: 2, 4, 8, 16, 32 and 64 was made with the following additional parameters:

N_tv	N_cbv	N_tv_for_cbv	dim	type_rng	factor_rng	factor_rng2	epsi	ft
2'048'00	2'048	1'000	12	3	500	5	0.001	0.01

Table 2: Used values for the simulations of section 5.3.

These simulations were made for the approach with decimation (version_HCB = 1) and for the approach with LBG (version_HCB = 3).

All simulations were done with the following computer: MacBook Pro (15-inch, end 2011), 2.5 GHz Intel Core i7, 8 GB 1333 MHz DDR3, AMD Radeon HD 6770M 1024 MB, OS X 10.8.5. The following MATLAB version was used: R2014a (8.3.0.532) 64-bit (maci64), February 11, 2014.

B Task Description

Institut für
Technische Informatik und
Kommunikationsnetze

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

(SA-2014-13)

Task description for the semester work
of
Mr. Tobias Bühler

Supervisor: Dr. Beat Pfister, ETZ D97.6
Tofigh Naghibi, ETZ D97.5

Issue Date: February 17, 2014
Submission Date: May 30, 2014

Speech Recognition with Hierarchical Codebook Search

Introduction

Statistical speech recognizers are commonly based on hidden Markov models (HMM). Most of them use so-called continuous observation densities (CDHMM). These CDHMMs fit well to the generally used speech features, i.e. the mel frequency cepstral coefficients (MFCCs). The time-dependent statistics of these continuous-valued features can efficiently be described with such CDHMMs.

For applications with limited proccessing power or even with integer arithmetic only, discrete density HMMs (DDHMMs) are much more desirable than CDHMMs. However, DDHMMs require discrete-valued features. Such features can be obtained from a vector quantization (VQ). A simple VQ with full codebook search needs also considerable processing power. Hence, only with the availability of an efficient VQ search the approach with DDHMMs may become a good solution.

The task of this semester thesis is to investigate an idea of efficient VQ that is based on a hierarchical search rather than on a full search. For that purpose we use a set of tree-like ordered codebooks as illustrated in Figure 1. The codebook at level 0 is given and has been optimized in some sense.[1] The codebook at the next higher level results from decimating the codebook at the current level. This is continued up to the root node.

[1]This optimization is not relevant here, because only the computational efficiency is considered.

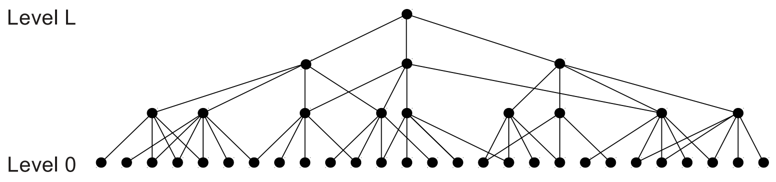

Figure 1: Search graph for a codebook of size 27 and a decimation factor of 3

Each node of the search graph has a number of daughter nodes that is higher than the decimation factor. We call this the splitting factor.

The important question in the context of this hierarchic approach is how to get the search graph. More specifically: How to attain an appropriate decimation method and how can the daughters of a node in the search graph be determined?

Constructing the search graph

The idea to construct the search graph for a computationally efficient VQ as sketched above is based on a set of hierarchically ordered codebooks. Given a codebook and a sufficiently high number of training vectors, a search graph with a decimation factor B can be constructed as follows:

1. The given set of M codebook vectors S_0 form the leaf nodes of the tree-like search graph. We term these nodes as level $l = 0$ of the graph and level $l = L$ forms the root. The number of nodes at level l is denominated as M_l and the corresponding set of nodes is S_l.
2. We move to the next level of the graph in direction to the root by incrementing l.
3. In order to get the $M_l = M_{l-1}/B$ nodes of level l, i.e. the node set S_l, we sequentially delete the nodes with the least increase of the distortion evaluated from the training vectors.
4. For each node j of level l we compute the frequency of the daughter nodes of level $l-1$ and store the set of daughter nodes with a frequency higher than some minimum value f_t as $D_{l,j}$. For this again the training vectors are used.
5. If M_l is greater than B we loop back to step 2. Otherwise the root of the search tree is added, i.e. the set of daughters of the root node $D_{L,1}$ is set to S_{L-1}, where $L = l+1$.

The height of the resulting search graph is determined by the selected value of the decimation factor B. The average branching factor depends on also on B, but additionally on the frequency threshold f_t.

The computational efficiency of the search graph as compared with the full search approach depends on the decimation factor as well as on the frequency threshold. The goodness of the VQ in terms of quantization loss depends solely on the frequency threshold.

Proposed workplan

It is recommended to proceed in this semester thesis as follows:

1. Read the chapters about linear prediction, feature extraction, vector quantization and HMM fundamentals in the textbook [1] and perform the associated laboratory exercises.

2. Look for the relevant literature and outline the basic approaches to hierarchical VQ (often also termed as tree-structured VQ). What have known approaches in common with the above sketched one and what are the differences?

3. Set up the detailed concept of the search graph construction in order to implement it in Matlab and estimate the required processing time as a function of the codebook size and the size of the training set.

4. Implement and test first a simplified approach: Instead of generating the hierarchical codebook by decimation, use the LBG algorithm (cf. [1], chapter 4.7) to generate it. Then determine the frequencies of the daughter nodes. Use the attained search graph in the VQ and test the program with suitable data, i.e. preferably synthetic low-dimensional test data.

5. If the estimated processing power (point 3) is within reasonable limits, implement the complete search graph construction in Matlab and test it with the same synthetic data. In case the required processing power is too high, discuss possible resorts with the supervisors.

6. Apply the VQ with search graph for a speech coding task as e.g. in the laboratory exercise number 10 of the speech processing lecture but with more speech data. Determine the efficiency gain as a function of the decimation factor and the frequency threshold. Also compute the quantization loss in function of the frequency threshold. Additionally test the subjectively heard speech quality as a function of the frequency threshold and evaluate the reduction of the required processing power.

7. Apply the VQ also in a speech recognition experiment like the laboratory exercise number 22 of the speech processing lecture. Compare the recognition rates for CDHMMs and DDHMMs.

The work done and the attained results have to be documented in a report (see recommendations [2]) that has to be handed in as PDF document. Furthermore, two presentations have to be given: the first one will take place some two weeks after the start of the work and is meant to give a short overview of the task and the initial planning. The second one at the end of the project is expected to present the task, the work done and the achieved results in a sufficiently detailed way. The dates of the presentations will be announced later.

References

[1] B. Pfister und T. Kaufmann. *Sprachverarbeitung: Grundlagen und Methoden der Sprachsynthese und Spracherkennung*. Springer Verlag (ISBN: 978-3-540-75909-6), 2008.

[2] B. Pfister. *Richtlinien für das Verfassen des Berichtes zu einer Semester- oder Master-Arbeit*. Institut TIK, ETH Zürich, Februar 2013.
(http://www.tik.ee.ethz.ch/spr/sada/richtlinien_bericht.pdf).

[3] B. Pfister. *Hinweise für die Präsentation der Semester- oder Master-Arbeit*. Institut TIK, ETH Zürich, Februar 2013.
(http://www.tik.ee.ethz.ch/spr/sada/hinweise_praesentation.pdf).

February 25, 2014